The ControlShell Component-Based Real-Time Programming System, and its Application to the Marsokhod Martian Rover

Stan Schneider Vincent Chen Jay Steele[1] Gerardo Pardo-Castellote

Real-Time Innovations, Inc.
954 Aster
Sunnyvale, California 94086

Recom Technologies
1777 Saratoga Avenue, Suite 206
San Jose, CA 95129

Abstract

Real-time system software is notoriously hard to share and reuse. This paper walks through the methodology and application of ControlShell, a component-based programming system for real-time system software development. ControlShell combines graphical system-building tools, an execution-time configuration manager, a real-time matrix package, and an object name service into an integrated development environment. It targets complex systems that require on-line reconfiguration and strategic control.

ControlShell takes advantage of functional object hierarchies to enable code sharing and reuse. It gains flexibility by supporting easy interconnectivity of these objects. It features a unique configuration control system for changing operating modes.

The paper concludes by examining the application of this framework to a teleoperated rover under development as a joint effort by NASA, several Russian space research institutes, and US industry. The rover is able to function remotely under control of a virtual reality interface.

1 Introduction

Motivation System programs for real-time command and control are, for the most part, custom software. Modern real-time operating systems [1,2,3,4] provide some basic building blocks—scheduling, communication, etc.—but do not encourage or enable any structure on the application software. Information binding and flow control, event responses, sampled-data interfaces, network connectivity, user interfaces, etc. are all left to the programmer. As a result, each real-time system rapidly becomes a custom software implementation. With so many unique interfaces, even simple modules cannot be shared or reused.

An effective real-time programming environment must facilitate sharing and reuse of program modules. It must assist the programmer both in structuring complex systems and in managing the system at run time. The framework must also provide services and tools to combine modules and build systems from reusable components. Finally, it must meet the many challenges unique to real-time computing, such as reacting to external temporal events, blending strategic-level command and low-level servo control, and switching between different modes of operation. All these challenges must be efficiently and smoothly handled by the architecture.

Summary This paper presents the rationale and motivation behind the *ControlShell* component-based programming system. ControlShell implements several new design concepts that have been proven effective in hard practice. For instance, while it takes advantage of object-oriented techniques, ControlShell differs from other object-oriented systems in that ControlShell objects implement functional units rather than model physical system components. ControlShell takes advantage of inheritance of these objects to provide complex functionality that is easily shared and reused. Also, these objects retain their identity in the run-time system, and are entered into an object name service. This allows unlimited interconnectivity.

ControlShell also addresses the fundamental issue of how to best merge event-driven reaction with feedback control. ControlShell presents a unique approach that uses an event-reaction programming system to change the data-flow pattern in a block-diagram model—without necessarily changing the diagram itself. This approach makes small and large changes in code configurations equally easy to implement. The result is a simple model that encourages fine mode control, and thus fine state definition, while providing considerable flexibility and generality.

This paper examines ControlShell, and illustrates its application to the Marsokhod rover project.

2 Approach

To our knowledge, ControlShell is the only framework combining component-based data-flow system construc-

1. Under Contract to the Intelligent Mechanisms Group of NASA-Ames Research Center, Moffett Field, CA

tion, event-driven state programming, a run-time executive, and transparent network connectivity.

ControlShell was developed to address a specific domain: complex electro-mechanical systems. It has been driven by practical applications from its inception [13,17,18,14, 15]. We start by describing the problem domain and objectives. We then examine ControlShell's architecture in the perspective of the many other approaches to real-time software development.

2.1 Strategic Objectives and Architectural Decisions

Practical Focus ControlShell is driven by practical, immediate considerations. It intentionally does not address design-time verification of real-time deadline constraints, nor guarantee that state machines will not deadlock. Instead, ControlShell assumes its target systems will be easily subjected to run-time validation both in simulation and on real hardware. Many tools are provided for addressing these issues at run time (execution profiling, flexible missed-deadline response, etc.). This trade-off has proven, in practice, to be very effective. For instance, it frees developers from making difficult estimates of execution times of complex code, and allows a more general state-programming model and complex state-transition action routines. We have chosen, essentially, practical flexibility over provable correctness, at the (minor) cost of extra run-time testing.

Reuse and Sharing ControlShell concentrates on code reusability and sharing. This is one of the factors that drove the development of the "functional" object hierarchies. By building functional units rather than modeling physical system components, ControlShell developers can take advantage of complex system building blocks that can be applied to many applications and physical systems. We thus strike a compromise between functional block-diagram tools (see below), and the power of object-behavior inheritance.

Programming System. ControlShell is designed, from the start, as a programming system. While many ControlShell applications can be built without custom code by linking pre-existing libraries, the emphasis has always been on providing a development environment that talented programmers will be happy to work with. ControlShell strives always to allow creative users to implement inventive solutions beyond the framework designer's original intent.

Separable services ControlShell is structured as a set of inter-related "services". ControlShell strives to provide tools that make sense in a complex system. Integration is accomplished by providing extensive, open interfaces. Since all interfaces between services are open; users may choose to replace almost any portion of the system with designs (or research results) more to their own liking. This decision has resulted in a flexible system that still works together (nearly) seamlessly.

Interconnectivity Complex systems often have inter-module interactions. For instance, an event-driven strategic control module must be able to interact with motion controllers, low-level routines must be able to interact with each other and raise conditions that higher levels handle. In a complex system, these interactions are often difficult to foresee or even characterize. ControlShell addresses this challenge by a) retaining the identity of all design-time objects in the run-time system, and b) providing a run-time "object name service", so any module may look-up any object at run time. This design provides very flexible connectivity.

Networking ControlShell is integrated with a network connectivity package called the Network Data Delivery Service (NDDS). NDDS is a novel network-transparent data-sharing system. It implements a "subscription" data-passing model that allows multiple clients to transparently and anonymously communicate data on a network. However, this functionality is beyond the scope of this paper, see [5,12] for details.

Data Acquisition ControlShell is also tightly integrated with RTI's *StethoScope* graphical data acquisition tool. StethoScope provides visibility and data collection for any variable in the ControlShell system. See [12] for details.

2.2 Perspective

There are many approaches to developing real-time system software, far too many to analyze them all here. Instead, we attempt to survey the general categories of tools, and differentiate their approaches from ControlShell's.

Hierarchy Specifications There are two quite different issues in real-time software system design: hierarchy (what is communicated), and superstructure (how it is communicated).

Several efforts are underway to define hierarchy specifications; NASREM[6] and UTAP[8] are notable examples. ControlShell makes no attempt to define hierarchical interfaces, but rather strives to provide a sufficiently generic software platform to allow the exploration of these issues.

Block Diagram Editors There are several functional block diagram editors and code generators. These include SystemBuild/AC100 by Integrated Systems Inc., and Sim-

ulink (a.k.a. the Real-Time Workshop) by The MathWorks, Inc.[11,3] These tools are heavily biased toward the low-end controls market. As such, they have interfaces for controls-design tools, and are powerful for choosing gains, designing controllers, etc. While they do have some facility for "custom" blocks, however, they are not designed as programming systems. Code generated from the block diagram combines both data objects and functional blocks into monolithic structures at run time. As a result, there's little interconnectivity and limited ability to develop complex, custom systems. These tools also do not address event-driven reactive programming.

Real-Time Formalism Tools There are several "traditional" real-time formalisms. Products exist on the market that implement some of these. For instance StateMate[7] by iLogix is based on Karel's StateCharts, ObjectTime is based on a Ward-Mellor variant called ROOM[22].

These systems concentrate on state machine behavior and interaction. They target a different application audience than ControlShell; they are mostly aimed at systems that are complex due to concurrency, such as a telecommunications system connected to 400 lines.

ObjectTime (ROOM) is object oriented. However, ROOM defines objects by modelling physical system components as state machines. A state machine in ROOM is a single object. This is fairly large-grained view of the world. In ControlShell, the state-transition routines are the objects. That allows users (even those working on very different applications) to develop reusable libraries of transition routine objects. Users can also inherit from base classes to construct complex action routines. This is advantageous in systems that perform non-trivial processing in the action routines. ObjectTime does not deal with data-flow or feedback configuration issues.

Onika Onika [23] is a "software composition system" from CMU. On the surface, it resembles ControlShell in that it composes systems from blocks of code. However, there are many major differences. Blocks in Onika are independent tasks. This forces a very "large-grained"' model, and is subject to loop delays. Onika supports simple changes in configurations (by substituting an entirely new diagram), but not graphical definition of complex configurations or event-driven reconfiguration. Onika does not address object-oriented issues, state programming, network connectivity, or automatic code generation.

Orcad/Esterel The Orcad system[10] provides an object-oriented design approach for robotic systems. Orcad combines control laws and reactive behaviors into objects called robot-tasks. Along with the Esterel language, Orcad strives for formal verification of temporal properties of control programs.

As in Onika, blocks (called module-tasks) in Orcad each run in a separate task context, and are thus large-grained. While state automata are provided, there is no notion of state programming, nor active transition routines. Operating mode switching is supported only by redefining the entire block diagram for each robot-task.

The next section analyzes ControlShell's system design methodology in the context of the issues presented above.

3 Methodology

3.1 Data-Flow Design Methodology

We term any system that has a periodic execution cycle a *data-flow system*. This includes most control loops, data acquisition systems, etc. These systems are sometimes also referred to as *sampled-data systems*.

Design Cycle The data-flow design process is shown in Figure 1. To design a data-flow system, the developer must first break the system into manageable components. Then each component is either implemented or selected from a library. The next step is to connect the components into an operational system. Testing the overall system provides the feedback required to drive a successful design.

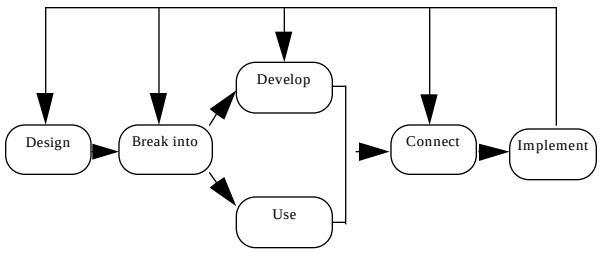

Figure 1. Data-Flow Design Cycle

ControlShell provides tools for every phase of the data-flow design cycle.

Components ControlShell builds data-flow systems from small, reusable objects called *components*. A component implements a specific functionality within a sampled-data environment via methods that run at well-defined times, such as at each sample-clock tick. By allowing components to attach easily to these critical times in the system, ControlShell defines an interface sufficient for installing (and therefore sharing) generic sampled-data programs.

Components read input signals, generate output signals, and use reference signals. Signals many be any of several types; most are named matrices called *CSMats*. Reference signals are often used for parameters, such as gains, names of other objects, or file names from which to load data.

Figure 2 shows an example of the *pdControl* component that implements a simple Proportional-Derivative controller.

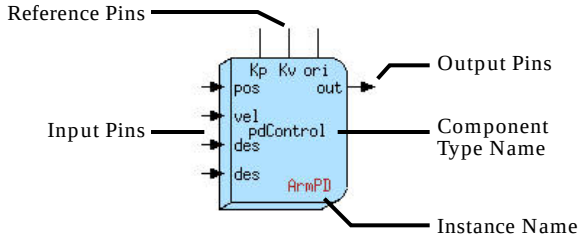

Figure 2. Example Component

Each component is labeled with a *type name* and an *instance name*. The instance names allow components to be reused in the same diagram. For example, the *pdControl* component's instance name in Figure 2 is *ArmPD*. Instance names are registered with *ControlShell*'s object name service. That way, other components and other *ControlShell* facilities—such as the finite-state machine—can bind to them at run-time and call their public methods, etc. This easy connectivity is a critical feature that allows *ControlShell* to support arbitrarily interconnected diagrams.

Each type of component is implemented as a C++ class. Components are derived either from a common base class, or from other components. Thus, components are built into class hierarchies of similar functionality. Derived components may hide (e.g. default) functionality or parameters to form an easier-to-use component, or add functionality to the base class, forming a more complex or functional component.

An example of an easier-to-use derived component is a Butterworth filter; it implements a simple type of filter derived from a generic filter class. It takes a few parameters (degree and cutoff frequency) instead of the direct filter coefficients.

An example of a more functional derived class is a component called *CmdNddsConsumer*. The CmdNddsConsumer component is from a base class that interfaces to NDDS. The base class subscribes to network data items and provides them for use by the data-flow system. The derived class also gets the data from the network, but also sends a stimulus to a state machine announcing data arrival (See Figure 3). These components provide a powerful and simple means of implementing network-distributed data flow and reactive behavior.

A library of pre-defined components is provided, ranging from hardware device drivers and controllers to trajectory generators and sophisticated motion planning modules. New or custom components are easily added to the system via a graphical data interchange editor and C++ code gen-

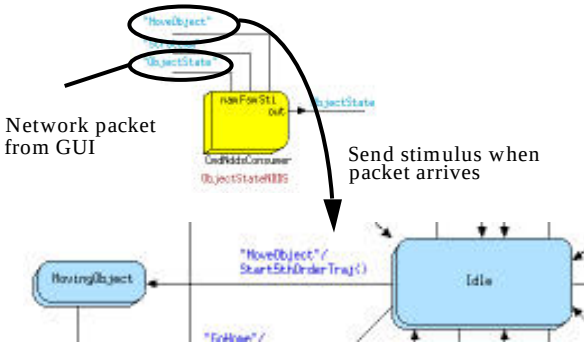

Figure 3. Functional Class Hierarchies

Sophisticated actions can be built from simpler functional classes. Here, the arrival of a network command causes a dual arm robotic system to move an object.

erator. This tool makes building and maintaining hierarchies of components simple to manage.

Connections Components are connected within a graphical tool called the *Data-Flow Editor* (DFE), shown in Figure 4. A system may be built from many separate block diagrams.

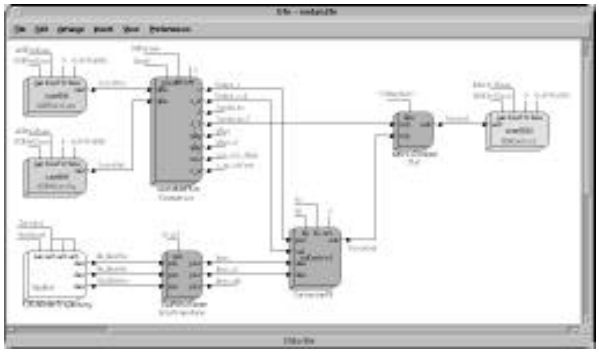

Figure 4. The Data-Flow Editor

The Data-Flow Editor connects components into systems. Blocks are components; lines on the diagram represent matrix objects. This diagram is a Cartesian-space controller for a 4-DOF SCARA robot.

Multiple diagrams are coordinated via the system manager. The system manager builds executing systems from sets of DFE diagrams. In Figure 4, two systems are set up. The system named "vx" will execute the actual hardware, the system named "unix" will execute the simulation. This capability makes hardware-in-the-loop simulation simple to set up. A similar setup lets the same high-level code run on different hardware configurations.

Execution The DFE outputs a textual language that describes all the connections in the system. The run-time executive parses the system description file, loads the required components, and dynamically links the signals

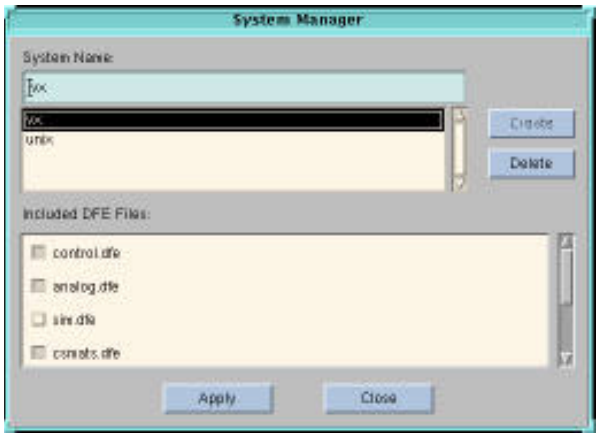

Figure 5. The DFE System Manager

The System Manager makes it easy to easily mix and match subsystems. In this case, it is being used to alternate between live execution and simulation.

specified in the block diagram (using the object service). New diagrams may be loaded at any time. Thus, ControlShell systems can be dynamically updated.

All component objects are placed on dynamic lists. The run-time executive orders the lists, thus scheduling the components' execution order to minimize delay. All components (that execute at the same sample rate) may then run as a single task (execution context). In multi-rate designs, a separate task is used to execute each sample rate present in the system. This design maintains each object's identity, while eliminating task-switch overhead between blocks.

Configurations Complex real-time systems often have to operate under many different conditions. The changing sets of conditions may require drastic changes in execution patterns. For example, a robotic system coming into contact with a hard surface may have to switch in a force control algorithm, along with its attendant sensor set, estimators, trajectory control routines, etc.

ControlShell's configuration manager directly supports this type of radical behavior change; it allows entire groups of modules to be quickly exchanged. Thus, different system personalities can be easily interchanged during execution. This is a great boon during development, when an application programmer may wish, for example, to quickly compare controllers. It is also of great utility in producing a multi-mode system design. By activating these changes from the state-machine facility (see below), the system is able to handle easily external events that cause major changes in system behavior.

Configurations are defined at design time by assigning components to groupings called *module groups* and *categories*, see [16,12] for details. System mode changes are then effected by the run-time configuration manager. It quickly reconfigures large numbers of active component objects, essentially redirecting the data-flow paths through the diagram.

This design offers much finer configuration control than other systems. By allowing the designer to implement many configurations on a single diagram, it eliminates the problems with maintaining several similar diagrams. It also encourages small changes in data-flow where appropriate. The named configurations are (of course) C++ objects, and are listed with the object service. Thus, they may be bound at run time by any module and activated when needed. The state programming system (discussed next) makes good use of this feature.

3.2 State Programming Design Methodology

All complex systems must be strategically guided. Since real-time systems must operate in a complex, event-rich environment, this means that the strategic control must react to many events. However, sequential processing is not well-suited to managing events. Event-driven programming—defining a sequence of events and the actions to take when the events occur—is much more appropriate. We term this *state programming*, because the process consists of identifying system states and the events recognized in those states.

Design Cycle The strategic control design process is shown in Figure 6. To design a strategic control system, the developer must first formalize the situation—identify the possible events the system may encounter, and specify what action the system should take in response to those events. The next step is to implement the action routines, or select them from a library. The final step is to connect the events to actions. Implementing and testing the design provides the feedback that makes the system work.

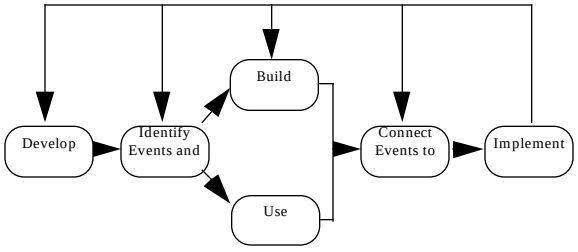

Figure 6. Strategic Control Design Cycle

ControlShell formalizes and assists reactive strategic programming.

To support strategic programming, *ControlShell* provides a state-machine programming system, consisting of a real-time state-machine engine, a graphical state-machine editor, and a state-transition-module (action routine) genera-

tion and management system.

Transition Modules The *ControlShell* state-programming system uses executable objects that implement actions in response to events. Because actions often result in state transitions, the objects are called *transition modules*. Transition modules implement a specific action, but are not intrinsically bound to an event. As with components, complex actions can be built by creating class hierarchies of transition modules.

Transition modules can accept parameters. For instance, one standard transition routine that activates a trajectory generator takes as parameters the name of the trajectory generator, the goal position, the slew time, and even the name of a configuration to activate before starting the trajectory. At run time, this transition routine will use the object data service to "hook-up" with the appropriate trajectory generators, configurations and data.

New transition modules are created via a graphical editor that defines the module's name, base class, formal parameter list, and possible return codes. A C++ code generator generates the code required to interface the new object to the system.

The ability to accept parameters combined with the ability to inherit the functions of existing transition modules makes transition modules easy to share and reuse.

Connections Transition modules are bound to events within the graphical State Programming Editor (SPE). Events in ControlShell are defined as boolean expressions of stimuli, where stimuli can be assertions (e.g. "Power = on") or triggers (e.g. "Contact"). Specifying a state transition therefore requires specifying a) a boolean expression (rule) that triggers the transition, b) the action (transition module) to execute, and c) the possible next states, depending on the return status of the transition module. All these are entered within the graphical SPE tool shown in Figure 4. The result is a graphical description of the events and actions required to complete a task: a state program.

In addition, the tool allows assigning values to the transition module's parameters. As with components, transition modules may use the object service to bind to any other object in the system. For example, a transition module that wants to take the action of moving a robot arm could look up and activate the control configuration that will drive the arm, and then find the trajectory generator that will cause the motion and start it. If this action is tied to the stimulus generated by the CmdNddsConsumer discussed above, it will allow a complex motion to occur in response to a network command. This easy integration results in considerable power.

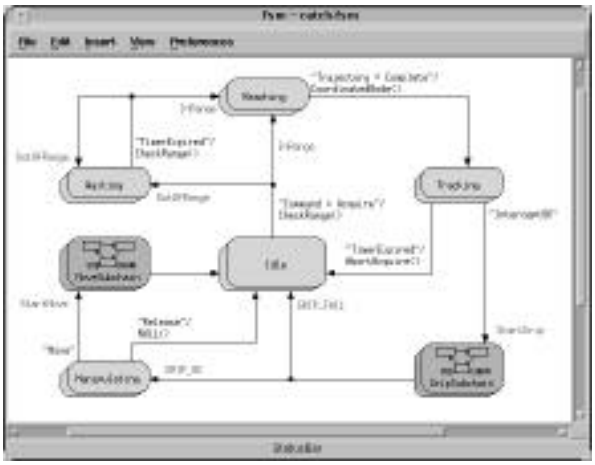

Figure 7. The State Programming Editor

Blocks in the SPE are states, the arrows represent transitions. The arrow labels are the boolean transition trigger expressions and transition module names. Transition parameters are set by clicking on the labels.

State Machine Engine The real-time state machine engine is designed to provide strategic control, while also managing concurrency in the system. ControlShell's state machine model features rule-based transition conditions, true callable subroutine hierarchies, task synchronization and event management. The details are beyond this paper, see [16,12]. It is, however, worth noting that the callable state subroutine concept also encourages reusability of state programs.

Execution As with DFE files, the SPE generates a textual language description of the state program. This description is parsed and linked by the run-time executive, and can be updated dynamically at run time. Each state program is executed by a separate task; stimuli are sent through message queues to the task.

4 The Marsokhod Rover

ControlShell has been used in many applications, including:

- Dual-arm cooperating robots
- Free flying space robotic systems
- Adaptive control (several systems)
- Cooperating mobile robot teams
- Underwater vehicle control
- Flexible structure control
- Mini-manipulator control
- Manufacturing workcell integration of planning and control
- Control of a 7-DOF Robotics Research arm

- Assembly of space structures with a Puma manipulator
- Wing rivet inspection

One of these systems is shown in Figure 8. The vehicle in

Figure 8. The Marsokhod Lunar/Martian Rover

This mobile robotic system is a Russian-designed platform, with NASA avionics and a ControlShell controller

the figure is the result of a collaboration between NASA's Ames Research Center (ARC), McDonnell Douglas Aerospace (MDA), NASA's Johnson Space Center (JSC), the University of Hawaii, and the Planetary Society (TPS), in addition to Russian's Lavochkin Association, VNIITransMash, and IKI. The project's immediate goal is to test the use of virtual-reality (VR) telepresence technology for controlling extra-terrestrial vehicles.

The exploration vehicle utilizes the six-wheeled Russian rover (Marsokhod) chassis, a MDA 5 DOF robotic arm, and the Virtual Environment Vehicle Interface (VEVI) VR software from Ames [24]. The vehicle carries stereo cameras for range mapping and uses a Global Positioning System (GPS) sensor, magnetic heading sensor, and inclinometers for terrestrial testing. For future extra-terrestrial simulations, the vehicle will carry laser mappers and accelerometers.

The vehicle controller integrates these systems and provides arm motions, low-level vehicle motion control, state estimation, remote operations, and the ability to execute paths through waypoints provided by the virtual reality interface.

For lunar operation, the vehicle will operate in a teleoperated mode, allowing a VR helmet-wearing operator to "drive" the vehicle almost as if driving a car. However, the long communications delays during Martian operation force more autonomy in the control system in order to increase mission productivity.

The Marsokhod vehicle is shared between several sites. An existing wheeled rover based on a power wheelchair frame (the Mobile Exploration Landrover, MEL) is used for building and testing of the controls software when the Marsokhod is unavailable. Code sharing and reusability, both between sites and vehicles, is therefore critical.

ControlShell's component-based design allows MEL and Marsokhod to share virtually all the data-flow and strategic-flow code. The design is a multi-rate, distributed application that integrates a graphical user interface running on UNIX workstations with strategic and low-level control running on the vehicle computer.

Wheel Control Figure 9 shows the vehicle wheel con-

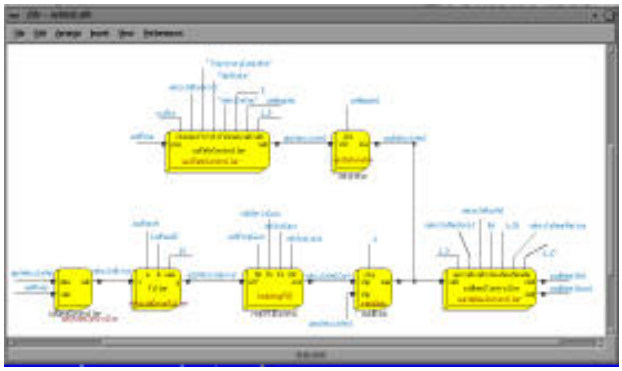

Figure 9. Wheel Control

This data-flow diagram implements the wheel-control logic. It allows both manual operation and autonomous path following.

troller for the rover. It implements two modes of control—a rate controller that accepts direct rate commands and a path controller that accepts a set of way points. Each controller computes the commanded vehicle velocity that the "ssWheelController" component (at the right of the figure) converts to commanded wheel velocities and accelerations for the hardware wheel-motor drivers (not shown).

ControlShell's configuration management system allows the system to switch control modes. For instance, Figure 10 shows the "RateControl" components. These components implement a velocity feedback loop with desired values from the network interface and the output of the pose estimator for a feedback signal. The "PathControl" mode is similarly defined.

The vehicle control takes advantage of several "off the shelf" components. For instance, the filter component was

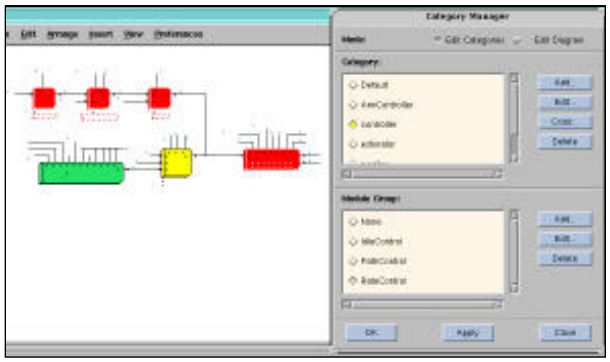

Figure 10. Rate Control Mode

Graphical mode definition makes it easy to support multiple modes. These blocks are enabled when rate-control mode is active. Trajectory generation, vehicle pose estimation, and sensor components are also switched when the mode changes.

convenient for constructing a low pass filter to filter out noisy encoder position data.

Other components are custom-generated for this project. For instance, some of the Marsokhod/MEL vehicle sensors (e.g., GPS sensor) provide very slow updates, while others—such as encoders—provide fast updates. ARC created a sophisticated, self-configuring Kalman Estimator that "fuses" the readings from all sensors in the multi-rate system to provide state estimation for the vehicles. ControlShell's C++ code-generation facility and component development system were used to integrate these custom components and device drivers with "standard" modules.

Strategic Control Figure 11 and Figure 12 show main features of the Marsokhod control strategy. Network command packets send stimuli to the finite-state machine, triggering changes in the low-level controller configuration and trajectory generation. The Idle states in both Figures represent the same state of the vehicle, where the vehicle wheel motors and the arm joint motors are not being commanded to move.

Figure 11 shows the strategic mode-control logic for the Marsokhod arm control. The primary driving factor behind the arm control strategy is to guarantee that the vehicle does not move while the arm is deployed. Vehicle motion on rough terrain produces vibrations that can be damaging to the deployed arm. Thus, only when the system is either in the Idle or JointIdle state will the controller respond to a "ArmJointMove" stimulus from the VEVI user interface, triggering the *ArmJointTrajectoryPlan()* transition routine and activating the ArmRateControlConfig configuration. And once the arm has been deployed, it is necessary to send a "ArmHome" stimulus in order to bring the arm back to a safe position before the controller is in the Idle state and ready to receive vehicle motion

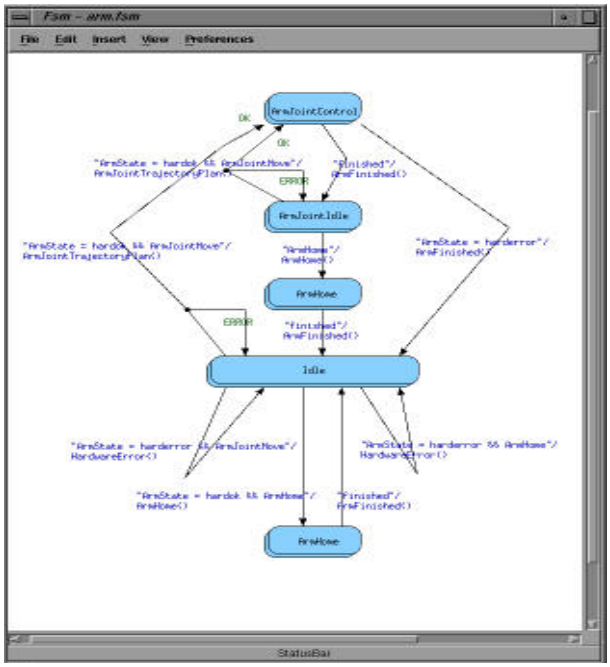

Figure 11. Arm Strategic Control

This state machine controls the operation of the vehicle, based on input from the network. Vehicle is commanded to move along a path or at a given rate.

commands. A hardware error in the arm joint motors naturally disrupts this strategy since it is no longer physically possible to "safe" the arm. In this case, the component overseeing arm joint motion control monitors stops the trajectory and sends a latched "VehState = harderror" stimulus which returns the controller state back to Idle. In this implementation, subsequent arm commands are matched with the condition "ArmState = harderror && ArmJointMove" and have no effect.

Figure 12 shows the strategic control over vehicle motion. For instance, a set of way points sent from the VEVI interface causes the "PathMove" stimulus. As illustrated in Figure 12, this triggers the *VehViaPtTrajectoryPlan()* transition routine to plan the trajectory. The transition routine also activates the "VehiclePathControlConfig" configuration; part of this configuration is the ssPathController component that relies on feedback from the state estimator to command vehicle velocity in order to reach each way point.

Remote commands also can switch the vehicle into rate control and change the arm controllers. The transition modules that provide configuration switching and trajectory activation are standard ControlShell modules. These modules may be inherited—according to C++ inheritance rules—to provide additional, customized capabilities.

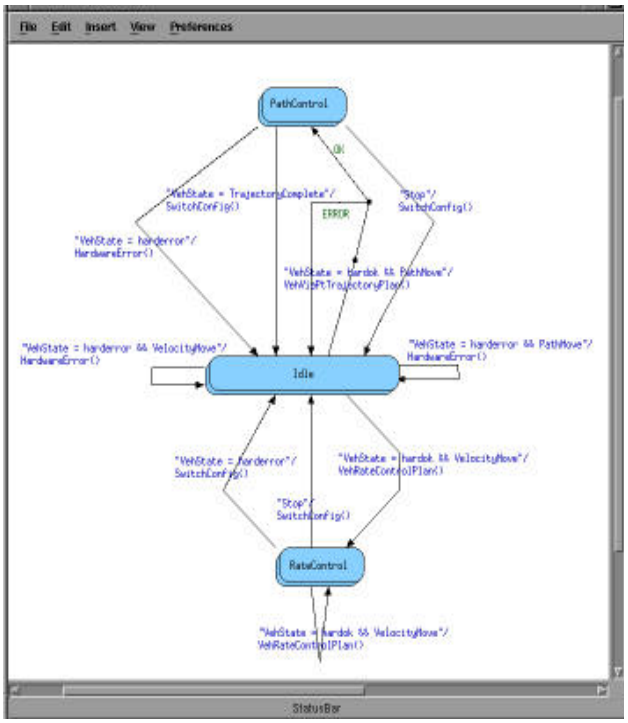

Figure 12. Vehicle Strategic Control

This state machine controls the operation of the vehicle, based on input from the network. Vehicle is commanded to move along a path or at a given rate.

Experimental Results ControlShell was fully implemented in ROM, and embedded in the Marsokhod vehicle for a full-system field test in Kilauea Volcano on Hawaii during February and March of 1995. This site was chosen for its varied geological features and difficult terrain, making it similar to that expected on a planetary surface.

All variables in ControlShell can be monitored with the StethoScope graphical display tool [12]. Figure 13 shows data taken from the actual vehicle operation, during a simple forward motion. Figure 14 illustrates heading control from actual vehicle operation, while Figure 15 shows path control of the vehicle in rough terrain with 4 way points.

5 Conclusions

This paper has presented a brief overview of the philosophies behind the ControlShell system. ControlShell is designed—first and foremost—to be an environment that enables the development of complex real-time systems. Emphasis, therefore, has been placed on a clean and open system structure, powerful system-building tools, and inter-project code sharing and reuse.

ControlShell is unique in offering:

- Functional object hierarchies, for *both* data-flow modules and action routines.

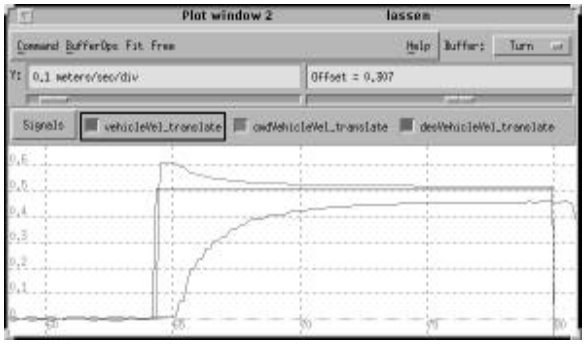

Figure 13. Vehicle Rate Control Performance

This plot shows a step in desired velocity, and the resulting feedback and estimated pose rate change. The estimator uses a slow time constant to insure a noise-free position signal. This results in the slow velocity rise time shown.

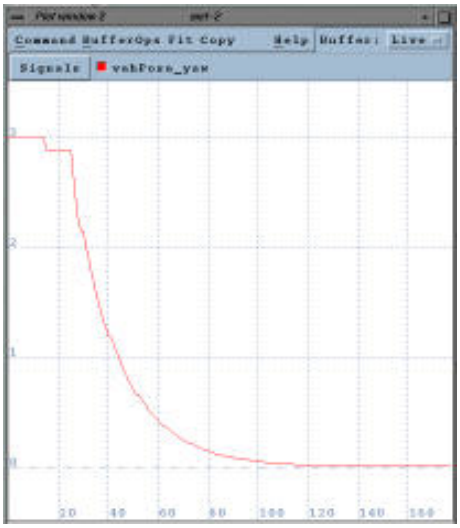

Figure 14. Vehicle Heading Control Performance

This plot shows actual vehicle heading control as the vehicle moves. In this example the actual heading has been artificially changed to 3.0 radians and the desired heading is 0.0 radians.

- Integrated state (strategic) and data-flow (servo-level) programming.
- Object persistence and an object name service, resulting in unlimited connectivity.
- Fine-grain blocks, executed in a single task context and ordered for minimal delay.
- Sophisticated operating mode (configuration) management.

ControlShell is in the process of being released as a commercial product. It has already found considerable application in universities, government, and industry. For instance, it has been embraced as the basis for much of the

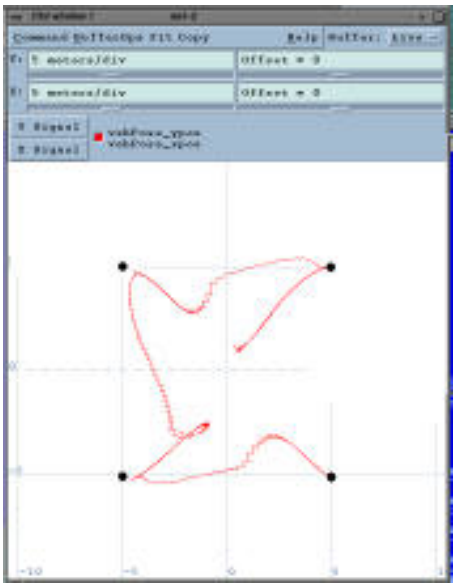

Figure 15. Vehicle Path Control Performance

This plot shows actual vehicle path control in rough terrain. The vehicle controller received 4 way points and successfully moved to each point. Note that loose soil and rocks prevented the vehicle from moving along a more optimal path.

new robotics development at NASA Ames Research Center and Jet Propulsion Laboratory.

Acknowledgments ControlShell is being jointly developed by Stanford University and Real-Time Innovations, Inc. Portions of this work were supported under ARPA contract. The authors wish to thank Dr. R. H. Cannon, Jr. for his guidance and leadership. The authors would also like to thank the many developers at Stanford, Loral, and NASA and other sites who have contributed ControlShell components.

References

1 D. B. Stewart, D. E. Schmitz, and P. Khosla, “The Chimera II: Real-Time Operating System for Advanced Sensor-Based Robotic Applications,” IEEE Transactions on Systems, Man, and Cybernetics, vol 22, no 6, pp1282-1295, December 1192.

2 Wind River Systems, Inc., 1351 Ocean Ave., Emeryville, CA 94608, VxWorks User's Manual, 1988-1993.

3 Integrated Systems, Inc., 2500 Mission College Boulevard, Santa Clara, CA 95054, ISI Product Literature, 1990-94.

4 Ready Systems, Inc., VRTX User's Manual, 1994.

5 G. Pardo-Castellote and S. A. Schneider. The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications. In Proceedings of the International Conference on Robotics and Automation, San Diego, CA, May 1994. IEEE, IEEE Computer Society.

6 Lumia, et. al., “NASREM Robot Control System Standard,” Robotics and Computer Integrated Manufacturing, vol. 6, no 4, 1989

7 Harel, David, et. al., “StateMate: A Working Environment for the Development of Complex Reactive Systems,”, IEE Transactions on Software Engineering, V16, n4, April 1990

8 M. Leahey, “Universal Telerobotics Architecture Project,” [reference will be found in before publication].

9 R. G. Simmons. “Structured Control for Autonomous Robots” IEEE Transactions on Robotics and Automation, 10(1), February 1994.

10 D. Simon, E. Coste-Maniere, Roger Pissard, “A Reactive Approach to Underwater-Vehicle Control: The Mixed ORCCAD/PIRAT Programming of the VORTEX Vehicle,” programme 4 - Robotique, Image Et Vision, Unite De Recherche - INRIA-SOPHIA ANTIPOLIS, Domaine de Voluceau Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, November 1992.

11 The MathWorks, Inc., Cochituate Place, 24 Prime Park Way, Natick MA 01760, Product Literature, 1990-93.

12 Real-Time Innovations, Inc. 954 Aster, Sunnyvale CA 94086, (408) 720-8312. Product Literature, 1991-95

13 S. Schneider, Experiments in the Dynamic and Strategic Control of Cooperating Manipulators. Ph.D. thesis, Stanford University, Stanford, CA 94305, September 1989.

14 S. Schneider and R. H. Cannon, “Object Impedance Control For Cooperative Manipulation: Theory and Experimental Results,” IEEE Journal of Robotics and Automation, vol. 8, June 1992.

15 S. A. Schneider and R. H. Cannon, “Experimental Object-level Strategic Control with Cooperating Manipulators,” The International Journal of Robotics Research, vol. 12, pp. 338--350, August 1993.

16 S. A. Schneider, V. Chen, and G. Pardo, “ControlShell: a Real-Time Software Framework,” AIAA Conference on Intelligent Robots in Field, Factory, Service and Space, March 1994

17 M. A. Ullman, Experiments in Autonomous Navigation and Control of Multi-Manipulator Free-Flying Space Robots. Ph.D. thesis, Stanford University, Stanford, CA 94305, March 1993.

18 V. W. Chen, Experiments in Adaptive Control of Multiple Cooperating Manipulators on a Free-Flying Space Robot. Ph.D. thesis, Stanford University, Stanford, CA 94305, December 1992.

19 G. Pardo-Castellote, T.-Y. Li, Y. Koga, R. H. C. Jr., J.-C. Latombe, and S. Schneider, “Experimental Integration of Planning in a Distributed Control System,” in Preprints of the Third International Symposium on Experimental Robotics, (Kyoto Japan), October 1993.

20 G. Pardo-Castellote, S. Schneider, and R. Cannon, “Robotic Workcell Manufacturing without Scheduling or Fixturing”, IEEE Conference on Robotics and Automation, May 1995

21 G. Pardo-Castellote, Experiments in the Integration of Planning and Control of a Dual-Arm Manufacturing Workcell. Ph.D. Thesis, Stanford University, Stanford CA 94305, 1995

22 B. Selic, G. Gullekson, and P. Ward, “Real-Time Object-Oriented Modeling,” Wiley and Sons, 1994

23 M. W. Gertz, D. B. Stewart, and P. K. Khosla, “A Software Architecture-Based Human-Machine Interface for Reconfigurable Sensor-Based Control Systems,” in Proc of 8th IEEE International Symposium on Intelligent Control, Chicago, Illinois, August 1993.

24 L. Piguet, T. Fong, B. Hine, and E. Nygren, “VEVI: A Virtual Reality Tool For Robotic Planetary Explorations”, Virtual Reality World ‘95, February 1995.